Available online at www.sciencedirect.com
ScienceDirect
Knowledge-Based Systems 20 (2007) 329–335
Knowledge-Based SYSTEMS
www.elsevier.com/locate/knosys

BitTableFI: An efficient mining frequent itemsets algorithm

Jie Dong, Min Han *

School of Electronic and Information Engineering, Dalian University of Technology, Dalian 116023, China

Received 25 March 2006; received in revised form 5 August 2006; accepted 15 August 2006
Available online 18 September 2006

Abstract

Mining frequent itemsets in transaction databases, time-series databases and many other kinds of databases is an important task and has been studied popularly in data mining research. The problem of mining frequent itemsets can be solved by constructing a candidate set of itemsets first, and then, identifying those itemsets that meet the frequent itemset requirement within this candidate set. Most of the previous research mainly focuses on pruning to reduce the candidate itemsets amounts and the times of scanning databases. However, many algorithms adopt an Apriori-like candidate itemsets generation and support count approach that is the most time-wasted process. To address this issue, the paper proposes an effective algorithm named as BitTableFI. In the algorithm, a special data structure BitTable is used horizontally and vertically to compress database for quick candidate itemsets generation and support count, respectively. The algorithm can also be used in many Apriori-like algorithms to improve the performance. Experiments with both synthetic and real databases show that BitTableFI outperforms Apriori and CBAR which uses ClusterTable for quick support count.

Keywords: Data mining; Frequent itemsets; BitTable; Database compressing

1. Introduction

The problem of finding association rules was first introduced by Agrawal et al. [1]. This problem aims at discovering unknown relationships, providing results that can be the basis of forecast and decision [16]. For example, supermarkets or catalog companies collect sales data from the sale orders. These orders usually consist of the sale date, the items in the transaction and customer-ID. Through getting and analyzing of association rules of these orders, a lot of valuable information such as the buying patterns of consumers can be inferred [5]. Finding frequent itemsets is the most fundamental and essential problem in finding association rules, from which association rules can be directly generated [12].

Finding frequent itemsets from large databases, by using support confidence based framework is not a trivial task [6]. The search space is exponential in the number of database attributes and with millions of database records the problem of I/O minimizations become paramount [15]. Due to its important and difficult, finding frequent itemsets attracts huge attention in numerous research communities, a lot of algorithms have been proposed in the different mining literature algorithm [4,7,8]. Agrawal and Srikant [2] proposed Apriori algorithm which traverses the search space in a pure breadth-first manner and finds support information by explicitly generating and counting each node. FP-growth algorithm [9] uses prefix-tree structure to mine frequent itemsets without generating candidates and scans the database only twice. Many of them are based on Apriori algorithm. Park et al. [13] proposes DHP algorithm using direct hashing and pruning. Holt and Chung [10] proposes IHP algorithm using inverted hashing and pruning. And Li [11] extends the Apriori algorithm with effective horizontal pruning technique. These algorithms mainly focus on pruning the candidate itemsets to reduce the candidate itemsets amount.

Pruning technique has greatly improved the performance of finding frequent itemsets, but the candidate itemsets generation and support count process of these Apriori-like algorithms is still the most time-wasted

* Corresponding author. Tel.: +86 411 84707847.
E-mail address: minhan@dlut.edu.cn (M. Han).

0950-7051/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2006.08.005

process. Ashrafi et al. [3] presents an algorithm using a partition table structure to compress the database. The algorithm performs the support count on partition table not on database and reconstructs the partition table each iteration. Tsay and Chiang [14] proposes CBAR algorithm that uses cluster table to load the databases into main memory. The algorithm's support count is performed on the cluster table and do not need to scan all the transactions stored in the cluster table, so the time of support count is saved.

In this paper, an algorithm named as BitTableFI is presented. The algorithm has significant difference from the Apriori and all other algorithms extended from Apriori. It compresses the database into BitTable, and with the special data structure, candidate itemsets generation and support count can be performed quickly.

The rest of paper is organized as follows: Section 2 is the background of finding frequent itemsets and problems of Apriori algorithm. And then, in Section 3 the algorithm is described mainly in three parts: BitTable structure, quick candidate itemsets generation and support count algorithm, some examples are also shown. In Section 4, experiment results on both synthetic and real databases are given, and Section 5 is the conclusions.

2. Preliminaries and background

Finding frequent itemsets can be formally stated as follows: let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of distinct literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A transaction T is said to support an itemset $X \subseteq I$ if it contains all items of X, i.e., $X \subseteq T$. The support of an itemset X is the number of transactions that support X. An itemset is frequent if its support is greater than or equal to a user-specified minimum support threshold, denoted *MinSup*.

Apriori is one of the most popular frequent itemsets mining algorithms which many algorithms base on. Given a user-specified *MinSup*, Apriori makes multiple passes over the database to find all frequent itemsets. In the first pass, Apriori scans the transaction database to count the support of each item and identify the frequent 1-itemsets marked as L_1. In a subsequent k-th pass, Apriori establishes a candidate set of frequent k-itemsets (which are itemsets of length k) marked as C_k from L_{k-1}. Two arbitrary L_{k-1} join each other, when their first $k-1$ items are identical. Then, the downward closure property is applied to reduce the number of candidates. This property refers to the fact that any subset of a frequent itemset must be frequent. Therefore, the process deletes all the k-itemsets whose subsets with length $k-1$ are not frequent. Next, the algorithm scans the entire transaction database to check whether each candidate k-itemset is frequent. An example on how Apriori discoveries frequent itemsets is shown in Fig. 1. The database in this example denoted D will be used in future examples.

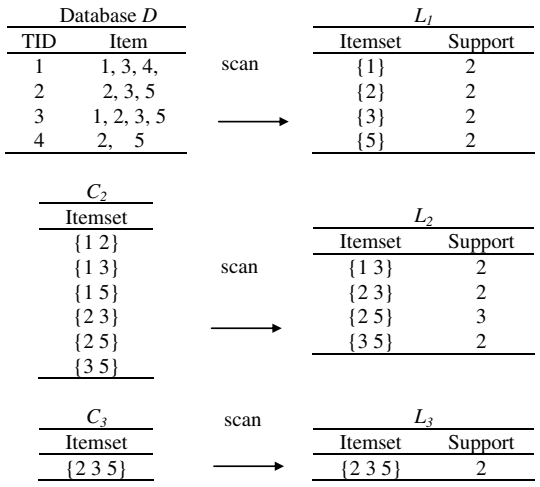

Fig. 1. Example of finding frequent itemsets.

There are two problems in Apriori algorithm. First, the generation process of candidate itemsets C_k need to compare all the items in each two itemset to check if they have the same first $k-1$ items and it employs a lot of CPU time. Second, the whole database needs to be scanned once to generate frequent itemsets L_k from C_k. In the scanning process, the algorithm count the times of each itemset in L_k appeared in the transaction. If L_k is generated, databases scan need to be performed k times. Reading disk resident data at every pass of the algorithm results large number of disk I/O operations. So, performance is dramatically affected, as the database is repeatedly scanned.

3. Algorithm descriptions

In this section, BitTable which the algorithm bases on to improve the performance is described first, and then the quick candidate itemsets generation and support count algorithm is given.

3.1. BitTable

BitTable is a set of integer whose every bit represents an item [5]. In BitTableFI, BitTable is used to compress the candidate itemsets and the database. For candidate itemsets compressing, if candidate itemset C contains item i, bit i of the BitTable's element is marked as one; otherwise, marked as zero. For database compressing, the BitTable is used vertically. If item i appears in transaction T, bit i of BitTable's T element is marked as one. If the size S of items or database transactions is greater than CPU's word size W, an array whose size is $S/W + 1$ is used to store the compressed data.

Example 1. Consider D of Fig. 1 and a candidate itemset $c = \{2\ 3\ 5\}$, c can be compressed to 13 whose binary code is 01101. And database D of Fig. 1 can be compressed to {10 7 14 8 7} for representation for {{1 0 1 0}, {0 1 1 1}, {1 1 1 0}, {1 0 0 0}, {0 1 1 1}} as shown in Table 1.

Table 1
Example of compressing database into BitTable

TID	Item 1	Item 2	Item 3	Item 4	Item 5
1	1	0	1	1	0
2	0	1	1	0	1
3	1	1	1	0	1
4	0	1	0	0	1
BitTable	10	7	14	8	7

3.2. Quick candidate itemsets generation

Before quick candidate itemsets generation can be performed, frequent 1-itemsets should be found first. All non-frequent 1-itemsets is neglected to reduce the size of BitTable and improve the performance of the generation process, because all non-frequent 1-itemsets is unuseful for further generation according to Agrawal and Srikant [2]. Then, frequent 1-itemsets are rearranged according to their lexicographic order and each item is marked with the sequence number as item ID. After frequent 2-itemsets was generated, frequent itemsets BitTable is constructed. Each element in BitTable represents a frequent itemset as described in Section 3.1. According to Apriori property [2], candidate itemsets C_{k+1} should be constructed by join two frequent itemsets of L_k that have common $k-1$ items. For each element E_1 in frequent itemsets BitTable BL_k, a middle variable *MID* is generated by replacing the last bit 1 of E_1 with 0. Each element E_2 after E_1 performs a Bitwise And operation denoted & with *MID*. If the result equals to *MID*, the two elements E_1 and E_2 in L_k have the common $k-1$ items. Then, a candidate itemset is generated by performing Bitwise Or operation denoted | on E_1 and E_2 and added to candidate itemsets BitTable BC_{k+1}. Compressing frequent itemsets into BitTable saves a lot of memory, and Bitwise And/Or operation is greatly faster than comparing each item in two frequent itemsets as described in [2]. The process of quick candidate itemsets generation is shown in Fig. 2.

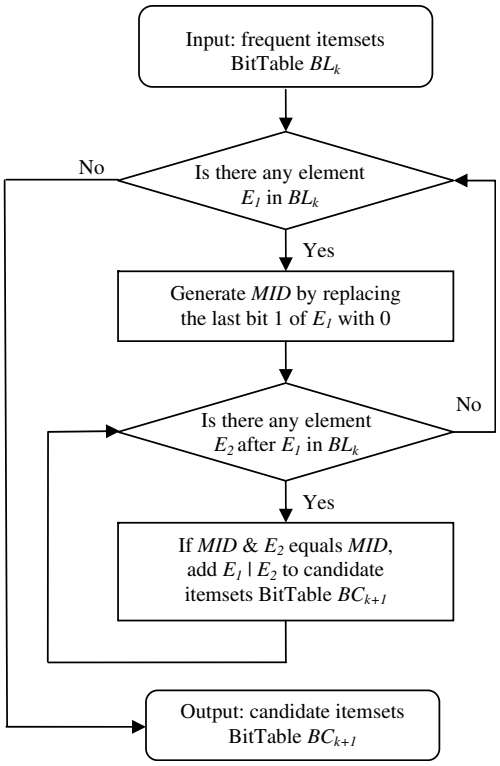

Fig. 2. The process of quick candidate itemsets generation.

Example 2. Consider D of Fig. 1 and suppose $MinSup = 2$. After the first scan of the whole database, $L_1 = \{1\ 2\ 3\ 5\}$ is generated. Item 4 is not frequent 1-itemset, so frequent 1-itemsets are rearranged and itemset {1 2 3 4} is used in the mining process instead of $L_1 = \{1\ 2\ 3\ 5\}$ to reduce the size of BitTable. When BitTableFI finished, frequent itemsets are reverted. From L_1 and C_1, $L_2 = \{\{1\ 3\}\ \{2\ 3\}\ \{2\ 4\}\ \{3\ 4\}\}$ can be generated. There are totally 4 items in frequent 1-itemsets, so the binary code length of frequent itemsets BitTable's element is 4. Frequent itemsets BitTable's binary codes are {{1 0 1 0}{0 1 1 0}{0 1 0 1}{0 0 1 1}}, so its corresponding BitTable BL_2 is {10 6 5 3}. Itemset {1 3}' *MID* is 8 whose binary code is {1 0 0 0} by replacing the last 1 of {1 0 1 0} with 0. Performing Bitwise And operation on {6 5 3} which represents {2 3} {2 4} {3 4}, respectively, only 0 is generated, so {1 3} {2 3}, {1 3} {2 4}, {1 3} {3 4} can not be jointed to generate candidate itemsets. Itemset {2 3}'s *MID* is 4. The result of performing Bitwise And operation on 5 which represents {2 4} equals the *MID*, so {2 3} {2 4} can be jointed. After performing Bitwise Or operation, 7 which represents {2 3 4} is generated and added to the candidate itemsets. The progress is shown in Fig. 3.

3.3. Quick candidate itemsets support count

An initialization of database BitTable should be accomplished first before quick candidate itemsets support count algorithm can be performed. The whole database is com-

L_2

Itemset	Support
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

→

$L_2[i]$,$L_2[j]$	BitTable BL_2
{1 3} {2 3}	10, 6
{1 3} {2 4}	10, 5
{1 3} {3 4}	10, 3
{2 3} {2 4}	6, 5
{2 3} {3 4}	6, 3
{2 4} {3 4}	5, 3

→

gen_ht($L_2[i]$)	*MID* & $L_2[j]$	BitTable BC_3
8	0	
8	0	
8	0	
4	4	7 {2,3,4}
4	0	
4	0	

Fig. 3. Example of quick candidate itemsets generation.

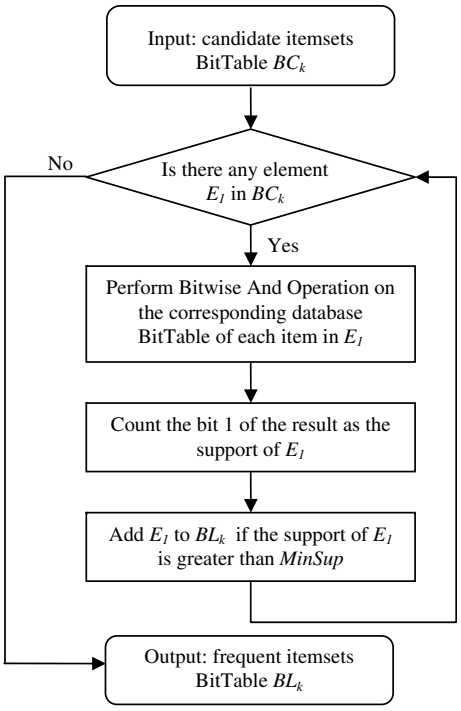

Fig. 4. The process of quick candidate itemsets support count.

pressed into a database BitTable so that it can even be stored in main memory. The database BitTable's compressing proportion is about (F1 Items/(CPU word size * All Items)). For example, if the database contains 1000 items, 32 k transactions and 200 frequent 1-itemsets, for 32 bit CPU, the compressed database BitTable's size is 1/160 of origin database. The initialization progress not only compresses the database, but also is the foundation of the quick candidate itemsets support count. The initialization progress of database BitTable acts as follows: for each item in frequent 1-itemsets, the binary values of the item in database transactions are combined to an integer which is added to the database BitTable later. If the size of the database transactions is larger than the CPU word size, an integer array is used.

The quick candidate itemsets support count algorithm counts the support of the candidate itemsets directly on the database BitTable. For every item in each candidate itemset, the element of the database BitTable corresponding to the item can be directly found out, because they have the same order as the frequent 1-itemsets. Performing Bitwise And operation on the corresponding elements of the items in each candidate itemset, the count of the bit 1 of the result is the support of the candidate itemset. If the support is greater than the *MinSup*, the candidate itemset becomes a frequent itemset. The process of quick candidate itemsets support count is shown in Fig. 4.

Example 3. In database D, $L_1 = \{1\ 2\ 3\ 5\}$. For item 1, its corresponding database value is {1 0 1 0}, so its database BitTable value is 10 whose binary value is 1010. After the initialization of the database BitTable, the database

item 2		item 3		item 5		support
0		1		0		0
1	&	1	&	1	→	1
1		1		1		1
1		0		1		0
7		14		7		6

Fig. 5. Example of candidate itemsets support count.

```
BitTableFI(Database D,int MinSup)
{
  L1 = { frequent 1-itemsets};
  BitTable[] db = database_bittable_init(D,L1);
  L2=getl2froml1(L1);
  int k=2;
  while (Lk.NotEmpty())
  {
    k++;
    Ck = quick_candidateitemsets_generation(Lk-1);
    Lk = quick_candidateitemsets_support_count(Ck, db, MinSup);
  }
}
```

Fig. 6. BitTableFI algorithm.

BitTable is compressed as {10 7 14 7}. When counting support for $C_3 = \{7\}$ representing {2 3 4}, 7 & 14 & 7 equals 6 representing {0 1 1 0}, so the support of candidate itemsets {2 3 4} is 2, which means {2 3 5} is frequent itemset, as shown in Fig. 5.

3.4. Algorithm description

Finally, the pseudo-code of BitTableFI is shown in Fig. 6. First, BitTableFI generates frequent 1-itemsets and initializes the database BitTable, and then uses quick candidate itemsets generation and quick candidate itemsets support count described above to generate all frequent itemsets until there is no itemset in L_k.

4. Experimental results

BitTableFI, along with Apriori and CBAR is implemented in C++ and compiled with Microsoft Visual C++ 6.0. All the experiments are performed on a 2.4 GHz Pentium4 PC with 768 MB memory, running Windows 2000. Total time, candidate itemsets generation time and support count time is compared, respectively. Two real databases used previously in the evaluation of frequent itemsets [5] and two synthetic databases which have

been proposed by Agrawal and Srikant [2], and used in [2,3,10,11,13] are used in the experiment. Two real databases' characteristics are shown in Table 2. Two experiment results on synthetic databases are shown in Figs. 7–12.

Figs. 7–9 show the total time, candidate itemsets generation time and support count time comparison results on database pumsb* among three algorithms. Figs. 10–12 are comparison results on database mushroom among three algorithm.

Two experiment results on synthetic databases are shown in Figs. 13 and 14. Fig. 13 is the total time of three algorithms on T40.I5.D10K with 100 items. In this database, the size of average transaction size and average maximal potentially frequent itemsets is set to 40 and 10, respectively, while the number of transactions in the database is set to 10K. Fig. 14 is the total time comparison result on database T40.I20.D10K. There are pretty long

Table 2
Database characteristics

Database	Items	Avg. length	Records
pumsb*	7117	50	49,046
mushroom	120	23	8124

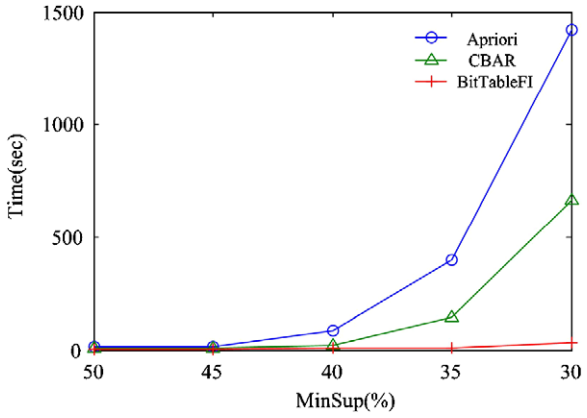

Fig. 7. Total time comparison on pumsb*.

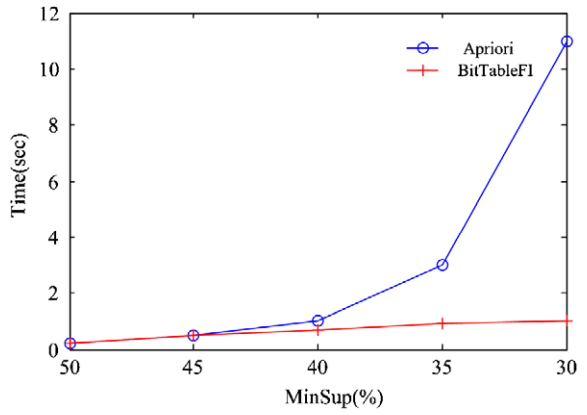

Fig. 8. Candidate itemsets generation time comparison on pumsb*.

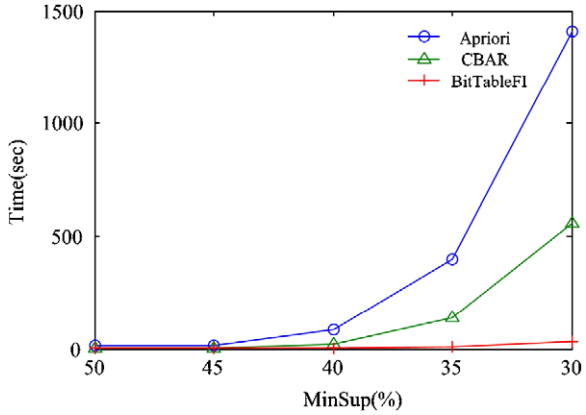

Fig. 9. Support count time comparison on pumsb*.

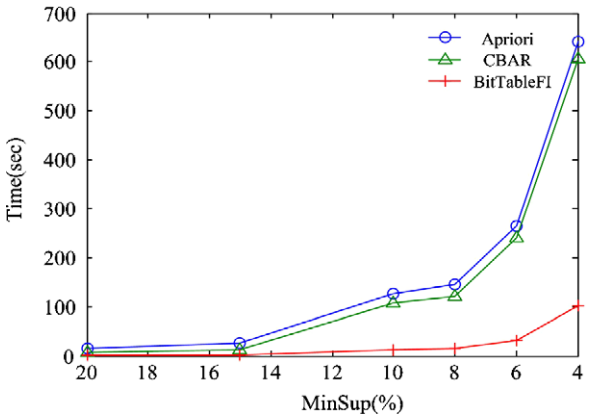

Fig. 10. Total time comparison on mushroom.

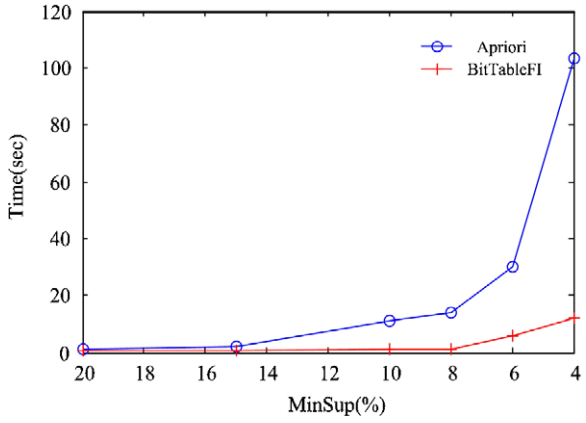

Fig. 11. Candidate itemsets generation time on mushroom.

frequent itemsets as well as a large number of short frequent itemsets in them.

The experimental results show that computation time sharply decreases with the decreasing of the MinSup in both Apriori and CBAR algorithm, while it increases slowly in BitTableFI. There is no candidate itemsets generation time comparison between CBAR and BitTableFI because CBAR uses the same algorithm as Apriori in candidate

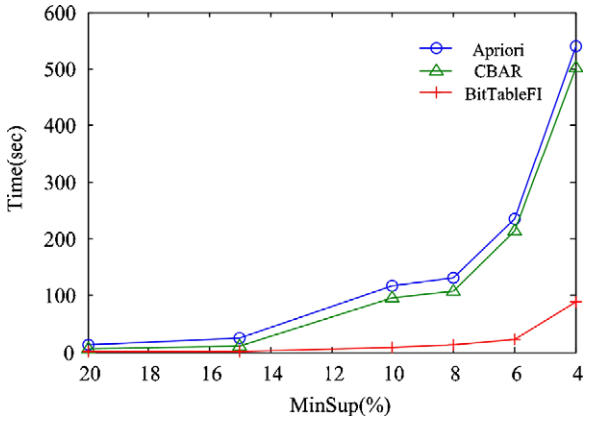

Fig. 12. Support count time on mushroom.

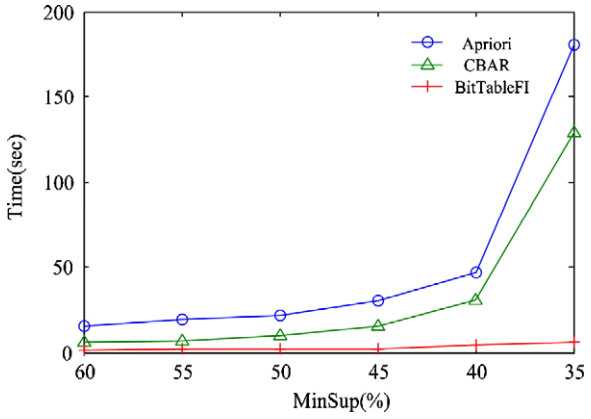

Fig. 13. Total time on T40.I5.D10K.

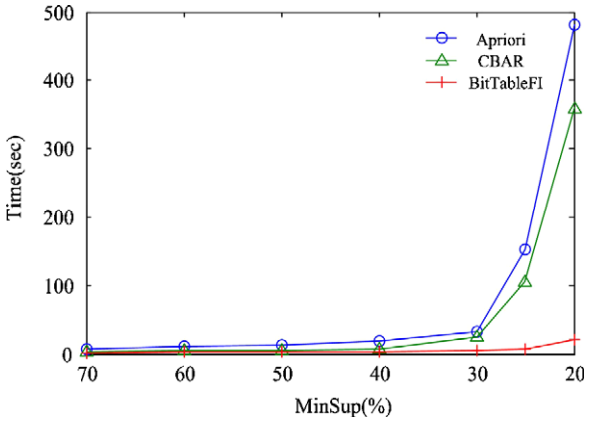

Fig. 14. Total time on T40.I20.D10K.

itemsets generation. BitTableFI outperforms Apriori and CBAR in both candidate itemsets generation and support count because of the special data structure and algorithm. So BitTableFI performs better than Apriori and CBAR in frequent itemsets mining.

5. Conclusion

In this paper, the problems of the Apriori-like algorithm are analyzed. To solve these problems, a special data structure, BitTable, for compressing the database and storing the itemsets is proposed and an algorithm, BitTableFI, for mining frequent itemsets from large databases is developed.

There are several advantages of BitTableFI over other algorithm: (1) it uses Bitwise And/Or operation to generate candidate itemsets based on BitTable which compresses the frequent and candidate itemsets. The Bitwise And/Or operation is greatly faster than the traditional item comparing method used in many Apriori-like algorithm. (2) It constructs a highly compact database BitTable, which is usually substantially smaller than the original database and the quick candidate itemsets support count algorithm can count the support of the candidate itemsets directly on it with Bitwise And operation. This saves the costly database scans in the subsequent mining processes. (3) The quick candidate itemsets generation and quick candidate itemsets support count technique can be easily used in Apriori-like algorithm. Fewer Apriori-like algorithms focus on these two most time-wasted problems.

However, BitTableFI focuses on solving the candidate itemsets generation and support count problems, so it does not take any pruning or other technique to reduce the size of candidate itemsets and the times of scanning the transaction database.

The BitTableFI is implemented and its performance in comparison with Apriori and CBAR on three aspects is studied, respectively. Four experimental evaluations on both synthetic data and real data show that BitTableFI algorithm outperforms those algorithms in most of cases.

References

[1] R. Agrawal, T. Imilienski, A. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, 1993, pp. 207–216.
[2] R. Agrawal, R. Srikant, Fast algorithm for mining association rules in large databases, in: Proceedings of 1994 International Conference on VLDB, 1994, pp. 487–499.
[3] M.Z. Ashrafi, D. Taniar, K. Smith, A compress-based association mining algorithm for large dataset, in: Proceedings of International Conference on Computational Science 2003, Lecture notes in computer science, vol. 2660, Springer, Berlin, 2003, pp. 978–987.
[4] F. Bonchi, F. Giannotti, A. Mazzanti, D. Pedreschi, Efficient breadth-first mining of frequent pattern with monotone constraints, Knowledge and Information Systems 8 (2) (2005) 131–153.
[5] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, T.M. Yiu, MAFIA: a maximal frequent itemset algorithm, IEEE Transactions on Knowledge and Data Engineering 17 (11) (2005) 1490–1504.
[6] K. Gouda, M.J. Zaki, GenMax: an efficient algorithm for mining maximal frequent itemsets, Data Mining and Knowledge Discovery 11 (3) (2005) 223–242.
[7] G. Goulbourne, F. Coenen, P. Leng, Algorithms for computing association rules using a partial-support tree, Knowledge-Based Systems 13 (2–3) (2000) 141–149.
[8] G. Grahne, J.F. Zhu, Fast algorithms for frequent itemset mining using FP-trees, IEEE Transactions on Knowledge and Data Engineering 17 (10) (2005) 1347–1362.
[9] J.W. Han, J. Pei, Y.W. Yin, R.Y. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, Data Mining and Knowledge Discovery 8 (1) (2004) 53–87.

[10] J.D. Holt, S.M. Chung, Mining association rules using inverted hashing and pruning, Information Processing Letters 83 (4) (2002) 211–220.
[11] Y.J. Li, P. Ning, X.S. Wang, S. Jajodia, Discovering calendar-based temporal association rules, Data and Knowledge Engineering 44 (2) (2003) 193–218.
[12] K. McGarry, A survey of interestingness measures for knowledge discovery, Knowledge Engineering Review 20 (1) (2005) 39–61.
[13] J.S. Park, M.S. Chen, P.S. Yu, Using a hash-based method with transaction trimming for mining association rules, IEEE Transactions on Knowledge and Data Engineering 9 (5) (1997) 813–825.
[14] Y.J. Tsay, J.Y. Chiang, CBAR: an efficient method for mining association rules, Knowledge-Based Systems 18 (2-3) (2005) 99–105.
[15] M.J. Zaki, Scalable algorithms for association mining, IEEE Transactions on Knowledge and Data Engineering 12 (3) (2000) 372–390.
[16] M.J. Zaki, Mining non-redundant association rules, Data Mining and Knowledge Discovery 9 (3) (2004) 223–248.